G4DB: A Scalable Datastore with Integrated Analytical Processing

Jeffrey Jedele
Technical University of Munich
jeffrey.jedele@tum.de

Xhens Basha
Technical University of Munich
xhens.basha@tum.de

ABSTRACT

G4DB is a distributed key-value-store with integrated data processing functionality. It is designed to store and analyze large amounts of unstructured data distributed across multiple servers. G4DB runs on commodity hardware where hardware-failures are the norm rather than the exception. It implements a peer-to-peer membership coordination mechanism to avoid the need for centralized coordination and is therefore scalable to a large number of nodes. An integrated MapReduce-engine makes it possible to run a wide range of analyses directly on the stored data.

1 MOTIVATION

In 2017, the total retail sales volume in the US equalled $3,496 billion [1] and it grows year by year. An ever increasing fraction of the total volume can be attributed to eCommerce. Amazon's Prime service has shipped 5 billion items in 2017 [5]. Alibaba reports to process 325,000 orders per second [11] in the same year.

Assuming an average order record is about 3kB in size[1], this translates into 1GB of new data being produced every second. Traditional relational databases that employ normalized storage schemes and provide advanced features like complex querying and multi-document transactions have problems scaling to such levels.

This problem gave rise to a new type of technology, the so-called *NoSQL* databases [18]. These leverage the fact that many types of data are accessed on document-level and a simple lookup by primary ID is sufficient. In return they make it easy to partition data across many nodes and are highly scalable by nature.

Storing such data often is a legal requirement, but it also is incredibly valuable. It is a vital ingredient to answer questions like which customers and products are most valuable to the business, how do customers navigate the site, how marketing campains should be designed, etc (cf. [7]). Some people nowadays go as far as calling data the oil of the 21st century.

Amazon e.g. is known to change product prices up to 2.5 million times a day, thereby increasing their sales by up to 25% [22]. It is however not only important that these questions can be answered, but they have to be answered as quickly as possible to reap the optimal benefits.

The work at hand presents *G4DB*, a highly scalable and failure-tolerant key-value-datastore with integrated MapReduce processing capabilities. It handles the data volumes present in current retail settings and allows users to run analyses directly on the stored data, thereby eliminating the latency involved by the need of moving it to external systems.

The structure of this report is as follows. Section 2 talks about related work which has influenced our design. Section 3 summarizes the key points of the G4DB design. Section 4 highlights some interesting implementation details. Section 5 presents some performance metrics validating our design. Section 6 finally concludes this work and introduces some avenues for future work on G4DB.

2 RELATED WORK

A significant amount of work has been done in the area of scalable key-value-stores. In 2006, a team of Google engineers designed *BigTable* [6], a distributed datastore to power their websearch. BigTable is built upon other services: data is stored in Google's distributed fileystem (GFS) and coordination is implemented using a highly-consistent external configuration service (Chubby). BigTable clusters are coordinated by a single master that has to be recovered in a time-consuming procedure in failure cases.

Dynamo [9] was designed by Amazon in 2007 as distributed datastore backing the shopping cart system. In contrast to BigTable, Dynamo is a self-contained system, i.e. it does not rely on other services. Instead of relying on a single master node for coordination, Dynamo implements a Gossip-based membership coordination mechanism. Dynamo aims at always accepting writes, at the cost of forcing readers to do conflict resolution. The idea is that users can easily remove things from their shopping carts again if they should appear due to conflicts.

Gossiping is a peer-to-peer (P2P) mechanism with the goal of broadcasting information to a number of nodes without the need of having pair-wise communication channels between nodes (which would imply factorial growth). Early work on using Gossip mechanisms for coordination of disitributed databases dates back to Alan Demers and colleagues in Xerox PARC in 1989 [10], but Dynamo is the first work to validate the approach with modern web-scale software to the authors' knowledge.

In 2010, *Cassandra* [19] has been developed at Facebook. Cassandra is very similar to Dynamo, but moves the responsibility of resolving conflicts back to the server using a simple last-write-wins (LWW) strategy. It has been open-sourced and there are many companies operating Cassandra clusters, one of the largest being Apple with 75,000 nodes and 10PB of data [12].

A major contribution to the field of distributed data processing was Google's works on *MapReduce* [8]. It has been found to provide a powerful abstraction for writing distributed and failure-tolerant data processing software. MapReduce has been implemented in many open source products such as Apache Hadoop [14] and Apache Spark [27]. It is one of the defining technologies for the "BigData" industry.

[1] Estimated based on the online retail data set described in [7].

Cloud Databases Lab Course (IN4163), WS 2018, Garching, Germany
2018.

There are some datastores with integrated MapReduce-engines, the best known ones likely being *MongoDB* [23] and *CouchDB* [13].

3 APPROACH

This section describes the cornerstones of G4DB's design with regards to scalability, failure-tolerance and efficient processing of large amounts of data.

The working assumptions are as follows:

(1) Both read and write workloads must be reasonably performant. Appending new data should not be unnecessarily blocked by current read operations.
(2) Data should not be lost, temporary inconsistencies for single items are OK.
(3) The system will encounter big and varying volumes of data and should be able to scale accordingly.
(4) With regard to failure-tolerance and scalability, the need for central coordinators should be reduced as much as possible.
(5) Data should not have to be moved to do analytical processing in order to reduce latency.

Due to the limited amount of space, we focus on the concepts of *partitioning*, *replication*, *membership coordination* and *MapReduce*-processing. The first three modules collaborate closely to handle read/write requests. A client initially connects to an arbitrary node in the G4DB cluster and sends requests to this node. The node knows about the whole cluster and determines if it is responsible for the request. If it is responsible, it persists the data and queues it up for replication. Replication happens completely asynchronously, so there can be periods of time where different replicas return different values for a key (eventual consistency). If it is not responsible, it sends the cluster information to the client. The client then redirects its request to the correct node.

3.1 Data Model

The data model of G4DB are simple key-value-tuples; both key and value are UTF-8-encoded strings. The client is responsible for encoding/decoding semi-structured data in an appropriate format like JavaScript Object Notation (JSON).

3.2 Partitioning

Scalability in distributed datastores is achieved by *partitioning* data across multiple nodes (sometimes also called *sharding*). G4DB's partitioning mechanism is based on *consistent hashing* [17]. Consistent hashing provides a mechanism to distribute a range of keys across a number of nodes such that only the close neighbors are impacted when we add or remove nodes. This is important since we want to scale G4DB clusters incrementally without having to move more data over the network than absolutely necessary.

The idea of consistent hashing is mapping both the set of node addresses $\mathcal{N}$ and the set of keys $\mathcal{K}$ onto the same linear space $\mathcal{H} = [min, max]$ using a hash function $h : \mathcal{N} \cup \mathcal{K} \mapsto \mathcal{H}$. This range is treated as a ring, i.e. the largest value wraps around to the smallest value. A node n is responsible for all keys which are hashed into the range $\text{resp}(n) = (h(n-1), h(n)]$, where n denotes the position on the ring (cf. Figure 1).

Consistent hashing in this basic form creates some challenges regarding uniform load distribution with a smaller number of nodes.

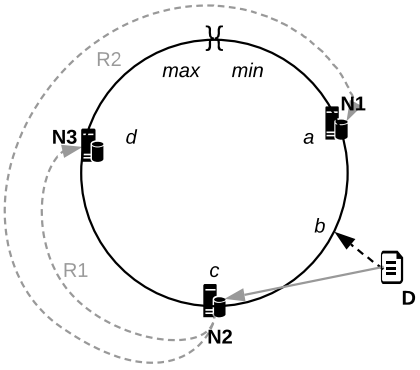

Figure 1: Consistent Hashing - Partitioning and Replication
Document D is written to the database and hashes to position b. It therefore lies in the responsibility of node $N2$, since $b \in \text{resp}(N2) = (a, c]$. With a replication factor of 3, D would also be replicated to nodes $N3$ and $N1$ which are the direct successors of $N2$.

One way to approach this would be representing each physial node by several virtual nodes that hash to different positions on the ring.

3.3 Replication

G4DB replicates data to 2 other nodes, thereby ensuring that two nodes can fail without data being lost. Consistent hashing provides a natural solution to coordinate replication: assuming a desired replication factor of 3, all keys that are stored on node n are replicated to nodes $n+1$ and $n+2$. The replication factor of 3 is currently fixed, but could be made configurable in future developments.

3.4 Membership

Each G4DB node must know all the other cluster nodes in order to determine for which data it is responsible so that it can handle requests correctly and communicate to clients where requests should go. When a node joins or leaves the cluster, the ranges of data for which its neighbors are responsible changes and must possibly be transferred from other nodes.

In a centralized implementation, an external configuration service (ECS) would be responsible for coordinating such processes. However this ECS would be a bottleneck given an increasing number of nodes it has to communicate with. P2P mechanisms like Gossip-broadcasting allow the nodes to handle this process amongst themselves in a more robust and scalable fashion. G4DB still uses an ECS as interface for system admninistrators, but this service is responsible only for bootstrapping and hands of further coordination work to the cluster as much as possible.

Gossip-based membership coordination works as follows. Each node in the cluster maintains its own view of the current cluster composition. The coordination takes place in rounds that are triggered by all nodes on a fixed schedule. During a round, the initiating node randomly chooses a small number of peers. It then exchanges its data with each of these peers, such that each party has the more recent data of the other party after the exchange. In the next round, new peers are chosen and the process is repeated with the updated

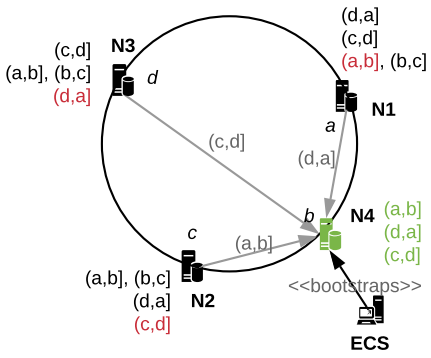

Figure 2: Membership - Adding a Node
Node 4 is added at ring position *b*. Next to a node are the key ranges in its responsibility, assuming a replication factor of 3. N4's synchronization engine will initiate data stream tasks (gray arrows) to obtain the green key ranges. Red ranges will be cleaned up at some point.

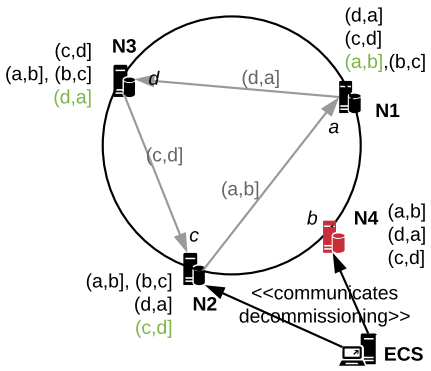

Figure 3: Membership - Removing a Node
Node 4 is removed again. The synchronization engines of all impacted nodes will initiate data stream tasks (gray arrows) to obtain the green key ranges.

state. With every round more nodes learn the updated information until at some point all nodes have received it.

The advantage of Gossiping is that it scales to a huge number of nodes without a factorial explosion in the number of communication paths. The disadvantage is that it is hard to reason about when all the nodes in the cluster will have converged to the same state.

Practical experiences with Dynamo and Cassandra have proven the suitability of this mechanism. We present some simulations for our implementation in Section 4.4.1.

3.4.1 Adding Nodes. Adding a node to the cluster is initiated via the ECS. The ECS connects to the physical node and starts a G4DB instance. The started instance receives the addresses of some other cluster members and starts Gossiping-rounds announcing its arrival. Two things happen in parallel: Firstly, each impacted predecessor switches the corresponding replication target to the new node. Secondly, the new node initiates data streams to obtain the data ranges for which it will be responsible after joining (cf. Figure 2). As soon a source node starts streaming, it activates a write lock to avoid data inconsistencies. After all data streams have completed, the new node changes its state to signal that it successfully joined the cluster. When other nodes receive this information, they reevaluate their responsibilities and start handling client requests accordingly.

3.4.2 Removing Nodes. The removal of a node can either be triggered manually by an administrator or by the failure detector when a node is believed to have failed. In both cases the process is triggered by the ECS. It changes the status of the node to decommissioned and seeds that status into the cluster. If the decommissioned node is still alive and receives this status update, it shuts down immediately. The neighbors of this node become responsible for additional data partitions due to the decommissioning. As soon they receive the updated status, they initiate the acquisition of such partitions which they do not already posses (cf. Figure 3).

G4DB it its current version uses the same mechanism to remove nodes, regardless if they are removed due to orderly downscaling or because of a failure-scenario. This means that even in the case of an orderly downscaling, the replication factor temporarily decreases. This could be avoided in future developments by implementing an orderly hand-off mechanism.

3.4.3 Failure Detection. G4DB's failure detection mechanism is currently implemented in a centralized fashion. The ECS continuously samples the cluster state from a small number of nodes and keeps track of the last time it has seen a status update for each node. Gossip messages contain a monotonically increasing heartbeat for this reason, and only a node itself will update its heartbeat. Therefore a node has likely crashed or is not reachable anymore if no messages with updated heartbeats disseminate through the cluster. If ECS does not observe updated heartbeats for a node for a specified amount of time, it triggers the decommissioning process for this node and then commissions a replacement.

This is a simple *accrual failure detection* mechanism which does not adapt itself to changing network conditions. Updating to the Phi Accrual failure detection mechanism described by Hayashibara and colleagues [15] could be a next step in the development of G4DB. Also the failure detection mechanism should be moved out of the ECS and into the cluster, e.g. by using a ring-based topology.

3.5 MapReduce

MapReduce is a programming model for data processing that originates from functional programming. It was discovered by Google engineers to provide a powerful abstraction for writing distributed and failure-tolerant data processing programs. The developer implementing a MapReduce-program specifies two functions:

```
map: (k1, v1) -> list((k2, v2))
reduce: (k2, list(v2)) -> v2
```

MapReduce is a batch-processing model. One processing stage proceeds as follows (cf. [20]), where meaninful processing pipelines can comprise several stages that build upon each others' results.

(1) The original data records are iterated over, the `map`-function is applied to each and produces one or more intermediate data tuples.

(2) The intermediate tuples are grouped by their respective keys.
(3) Each group is reduced to a single, aggregated value using the `reduce`-function.

It is important to note that the `map`-operation can be parallelized completely while the `reduce`-operation can be parallelized for different keys.

3.5.1 Topology. All relevant parts of a MapReduce-process happen on server-side, the client is not involved after a job has been started. A process comprises one master for the whole cluster and one worker per node.

3.5.2 Interaction. The client triggers a MapReduce-job via the API. The request contains the user script defining the `map`- and `reduce`-functions. The node which receives this script starts a master process. The master process creates a plan defining which node will process which data partition and federates the job request accordingly. Typically each node will process its primary partition, but this could differ in cases of fail-over. Each node starts a worker process when it receives the federated request. The worker process starts applying the user script on the data range specified in the request. Each worker eventually sends back its pre-reduced results to the master process. The master process executes one additional reduction upon receiving each worker result in order to combine them. The final result is stored in the cluster using the normal key distribution strategy.

3.5.3 Worker Fail-Over. If the master process does not receive the result from a worker within a certain amount of time, it will poll this worker to check if it is still processing data. If the polled node encountered an error or is not reachable, the master retriggers the processing for the corresponding data partition on one of the replica nodes.

It shall be mentioned at this point that the master process is a single point of failure and bottleneck in the current implementation. We will talk more about this in Section 6.

4 IMPLEMENTATION

4.1 Architecture

The implementation of G4DB conceptually comprises several high-level modules (cf. Figure 4). Their responsibilities are as follows.

Connection Acceptor Handles all incoming network communication from clients and peer nodes.
Data Request Handler Handles `PUT`/`GET` requests and takes care of replication.
Map/Reduce Request Handler Handles all MapReduce-related requests, initiates and maintains master and worker processes.
Admin Request Handler Handles administrative requests from the ECS.
Gossiper Executes the Gossip-based membership coordination.
Synchronization Engine Coordinates the reallocation of data when nodes join or leave the cluster.
CleanUp Engine Cleans up data for which a node lost responsibility due to cluster changes.
Persistence Service Responsible for data persistence and caching.

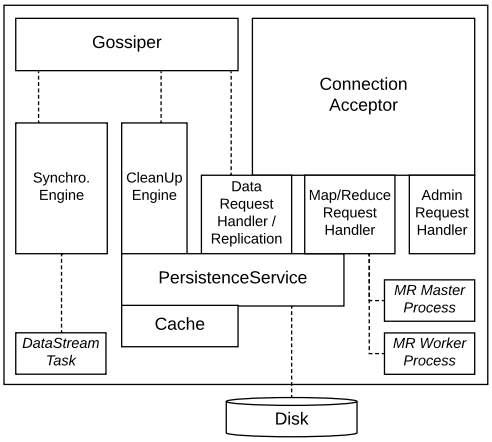

Figure 4: Architectural Overview

4.2 Local Persistence

The persistence layer of G4DB stores each key-value-pair in a separate file. This allows accessing data with minimal synchronization costs. Data is cached for reading, cache size and displacement strategy are configurable. G4DB provides first-in-first-out (FIFO), least-frequently-used (LFU) and least-recently-used (LRU) caches. All cache implementations provide $O(1)$ performance for both store and read operations [21, 26]. Writes are not cached and write operations are only considered successful when the data has been written to disk. This limits write performance, but ensures that successful writes are not lost when a node crashes. We will talk about approaches to improve the performance of the persistence layer in Section 6.

4.3 Network Protocol

The wire protocol of G4DB is ASCII-based. This induces some overhead compared to a binary protocol both in processing time and network traffic, but we deem this overhead acceptable for our usecase (cf. [3]). The benefits are less complex parsing logic and simplified debugging.

4.4 Gossiping

4.4.1 Peer Selection. A G4DB node initiates a Gossiping-round every second and chooses between 1 to 3 peers using the following process:

(1) One random node is chosen from the whole cluster.
(2) One of the seed nodes is added probabilistically with a chance of 30%. This aims at promoting the seed nodes to hotspots for the distribution of recent updates and thereby reducing dissemination time.
(3) One of the nodes believed as dead, i.e. one that has not replied in an earlier round, is added probabilistically with a chance of 30%. This aims at reevaluating node states timely after communication problems.

We have conducted a simulation to get a feeling for how many rounds of gossip it takes for a message to disseminate through the

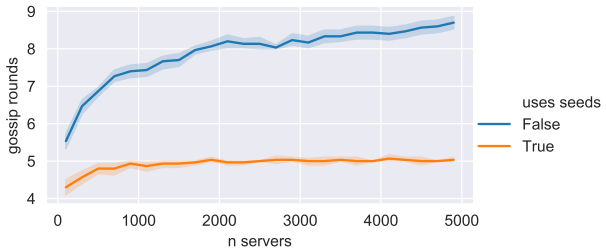

Figure 5: Scalability of Gossip-Communication
Number of Gossiping-rounds necessary for a message to completely disseminate through the cluster dependent on the number of nodes. In the first experiment, 1 peer is chosen in each gossiping round (blue plot). In the second experiment, one of 3 seed nodes is added probabilistically with a chance of 30% to the chosen random node (orange plot). Shaded areas denote 95% confidence intervals.

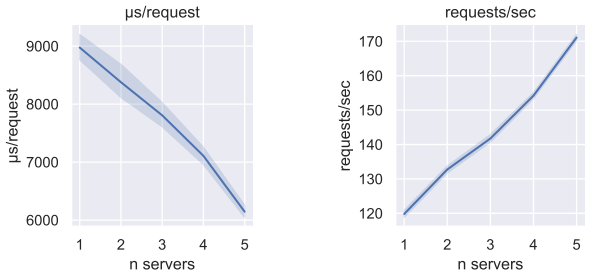

Figure 6: Latency-related Performance Characteristics
Payload size is 10kB. Shaded areas indicate 95% confidence intervals.

cluster. We have found that this time stays in reasonable bounds even for larger number of nodes (cf. Figure 5).

4.4.2 Message Structure. The structure of the Gossip messages is modeled after Cassandra (cf. [4]). It contains a status for each (known) node in the cluster. A node status comprises following fields:

Field	Description
Generation	Local timestamp set when a node starts.
Heartbeat	Updated regularly by the node.
State	One of `STOPPED`, `OK`, `REBALANCING`, `JOINING`, `DECOMMISSIONED`.
State Version	Counter that is updated every time the state is updated.

4.4.3 Communication Pattern. A Gossiping-round comprises a single request-response cycle between the initiator and each of its chosen peers. The initiator sends its full view of the cluster to the peer, the peer merges it with its own state and sends the updated state back to the initiator. The initiator than merges the received state back into its own state. Compared to Cassandra's three-way communication pattern this produces more network traffic since the full cluster state is send along with each message. As a benefit the implementation is more straightforward.

4.4.4 State Merging. During a Gossip-exchange both peers will typically have data about the same nodes. The merged state should only contain the more recent information. Recency is determined by state attributes in following order:

(1) *Generation*: If one entry has a newer generation time, this means the node has been restarted and all old information is obsolete.
(2) *State version*: The main reason to check the state version is node decommissioning. The ECS seeds an updated node state with state `DECOMMISSIONED` and incremented state version into the cluster. If the impacted node is still alive and receives such a message, it will take on the decommissioned status and shutdown immediately.
(3) *Heartbeat* All other things being equal, the state with the more recent heartbeat wins.

4.5 MapReduce

4.5.1 Master Placement. The MapReduce-API requires the client to come up with a unique ID. This allows G4DB to deterministically associate a MapReduce-job with a certain cluster node using the same consistent hashing mechanism that is used for data records. Like this, masters can be distributed across the cluster while the client remains capable of locating it, e.g. for polling the status after reconnecting.

4.5.2 Processing. Mapping and reducing are not executed sequentially but in a pipelined fashion in order to reduce memory consumption on the nodes. More specifically, there is a configurable buffer size per key. Once the number of mapped records exceeds this buffer size, a reduce operation is executed. After all records have been mapped, a final worker-side reduce is executed in order to have only one result per key. These results are then sent back to the master where results of different workers are combined using the same reduction logic.

The shuffle-phase happens implicitely in memory while mapped records are encountered. There is currently no mechanism to spill intermediate data to disk, which means that MapReduce-programs generating a large number of mapped keys will cause memory problems.

4.5.3 Scripting. The MapReduce-engine employs the Nashorn JavaScript engine [25] included with the JDK for user scripts. It implements the ECMAScript 5.1 [16] standard including additional extensions (cf. [2]).

5 EVALUATION

5.1 Read/Write Performance

We conducted the performance measurements for G4DB on a Google Cloud cluster with 5 nodes as servers and 3 nodes as clients. The nodes are n1-standard-1 instances with one 2.6GHz Xeon Core, 4GB RAM, 1000MBit network and non-SSD disks. As general setup for latency- and throughput-related performance measurements

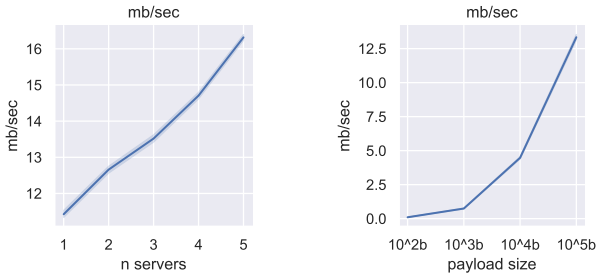

Figure 7: Throughput-related performance characteristics
Payload size in the left chart is 10kB. Shaded areas indicate 95% confidence intervals.

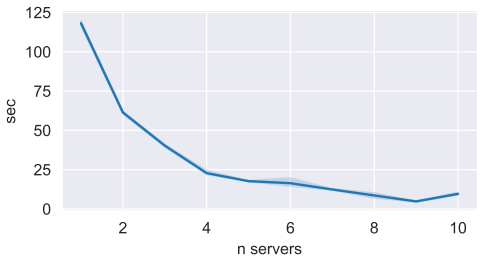

Figure 8: Performance of MapReduce
Time necessary to aggregate sales by country for 397,000 records. Shaded areas denote 95% confidence intervals. The fluctuation at the end hints at a situation where the data a single node has to process becomes so small that the network communication overhead exceeds the actual processing time.

we have 3 clients writing synthetic records of fixed size to a G4DB cluster with a varying number of nodes.

We find that write performance increases linearly with the number of server nodes (cf. Figure 6). This validates G4DB's design for scalability. As another interesting insight, we find that the throughput increases significantly when we increase message payload (cf. Figure 7). There are two likely explanations for this: Firstly, increased message payload reduces protocol overhead. Secondly, our persistence implementation currently is unbuffered and involves random rather than sequential disk access, which makes write operations expensive. We talk about avenues for improvement in Section 6.

We do not talk about read-performance in this report, since this is a more complex topic which depends on use case and cache configuration. We found the read-performance to be between 5-10MB/s given 10kB messages.

5.2 MapReduce Performance

To evaluate our MapReduce-implementation, we look at a simple use case of aggregating sales volume by country. We use the online retail data set compiled by Daqing Chen and colleagues [7]. We conducted minimal preprocessing to transform the data set from normalized CSV-format to a JSON-encoded version more suitable for a key-value-store. We also duplicated it 15 times to reach a more significant size of 397,000 records (1.9GB).

We had to slightly manipulate G4DB to simulate an approximately uniform distribution of data across nodes, but have then be able to process the data in approximately 5s with 10 nodes (cf. Figure 8).

6 CONCLUSION

We have designed and implemented G4DB, a highly scalable, failure-tolerant and widely applicable key-value-datastore with integrated MapReduce-functionality.

The current state of implementation should be seen as a proof-of-concept and there are many areas future developments could improve upon.

In Section 5 we already hinted at the inefficiencies of the currently rather primitive implementations of network protocol and persistence layer.

The protocol could be replaced with a optimized binary version to reduce some network traffic and decoding overhead. Since G4DB is designed to work with textual data, the savings by doing so should be expected to be rather marginal. More interesting avenues for the network communication could be adding compression and bulk requests that reduce network overhead.

The disk persistence currently uses one file per value and does not buffer writes to disk, i.e. a write is only reported as successful when the file has been written to disk. While this reduces the chances for data loss in case of failures, it is very inefficient. Huge benefits could be gained by employing a combination of append-only write-ahead-log (WAL) and log-structured-merge-trees (LSMs) [24].

The current MapReduce-implementation has a couple of scalability-related problems. Right now, the master process is responsible both for federating the original request to all nodes in the cluster as well as for consolidating the responses of all nodes. This makes it a bottle neck.

Instead of letting the master federate the request to all nodes, it could make sense to employ the Gossiping-mechanism. However it is not a good choice to distribute the user scripts, since including them in all Gossip-communication would blow up the network traffic significantly. An idea could be making each server that was "infected" by the message pull the script from the server it received the information from and then immediatly make it serve requests on its own. Secondly, a pruning mechanism for the Gossip mechanism should be implemented if it is used to exchange more information, such that outdated information does not accumulate.

For the consolidation of the worker results, a better approach would be letting each node process the results for the keys it has to store in the end. Challenges that come with this are making sure that replication is taken care of and additional status communication to detect when a job has finished.

REFERENCES

[1] Fareeha Ali. 2018. A decade in review: E-commerce sales vs. retail sales 2007-2017. Retrieved 2019-01-20 from https://www.digitalcommerce360.com/article/e-commerce-sales-retail-sales-ten-year-review/

[2] Sundararajan Athijegannathan. 2016. OpenJDK Wiki - Nashorn extensions. Retrieved 2019-02-02 from https://wiki.openjdk.java.net/display/Nashorn/Nashorn+extensions

[3] Kalid Azad. 2017. A little diddy about binary file formats. Retrieved 2019-01-26 from https://betterexplained.com/articles/a-little-diddy-about-binary-file-formats/

[4] Jason Brown. 2014. Demystifying Gossip. Retrieved March 21, 2008 from https://www.youtube.com/watch?v=FuP1Fvrv6ZQ

[5] Ashley Carman. 2018. Amazon shipped over 5 billion items worldwide through Prime in 2017. Retrieved 2019-01-20 from https://www.theverge.com/2018/1/2/16841786/amazon-prime-2017-users-ship-five-billion

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. USENIX Association, Seattle, WA. https://www.usenix.org/conference/osdi-06/bigtable-distributed-storage-system-structured-data

[7] Daqing Chen, Sai Laing Sain, and Kun Guo. 2012. Data mining for the online retail industry: A case study of RFM model-based customer segmentation using data mining. *Journal of Database Marketing and Customer Strategy Management* 19, 3 (Aug. 2012), 197–208. http://researchopen.lsbu.ac.uk/1492/

[8] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 205–220. https://doi.org/10.1145/1323293.1294281

[10] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC '87)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/41840.41841

[11] Jessica Fenol. 2018. Alibaba: 325,000 orders per second highlights e-commerce shift. Retrieved 2019-01-20 from https://news.abs-cbn.com/business/09/04/18/alibaba-325000-orders-per-second-highlights-e-commerce-shift

[12] Apache Software Foundation. 2016. Cassandra. Retrieved 2019-02-04 from http://cassandra.apache.org

[13] Apache Software Foundation. 2018. CouchDB. Retrieved 2019-01-20 from http://couchdb.apache.org

[14] Apache Software Foundation. 2018. Hadoop. Retrieved 2019-01-20 from https://hadoop.apache.org

[15] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. 2004. The /spl phi/ accrual failure detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.* 66–78. https://doi.org/10.1109/RELDIS.2004.1353004

[16] Ecma International. 2011. Standard ECMA-262 5.1 Edition. Retrieved 2019-02-02 from https://www.ecma-international.org/ecma-262/5.1/

[17] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC '97)*. ACM, New York, NY, USA, 654–663. https://doi.org/10.1145/258533.258660

[18] Martin Kleppmann. 2018. *Designing Data-Intensive Applications.* O'Reilly Media, Inc., Sebastopol, CA, USA.

[19] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[20] Ralf Lämmel. 2008. Google's MapReduce programming model — Revisited. *Science of Computer Programming* 70, 1 (2008), 1–30. https://doi.org/10.1016/j.scico.2007.07.001

[21] Kendeh Lewis Malike. 2017. Creating An In-Memory FIFO Cache. Retrieved 2019-01-26 from https://malike.github.io/Queue-LinkedHashMap.html

[22] Neel Mehta, Parth Detroja, and Aditya Agashe. 2018. Amazon changes prices on its products about every 10 minutes — here's how and why they do it. Retrieved 2019-01-20 from https://www.businessinsider.de/amazon-price-changes-2018-8

[23] Inc. MongoDB. 2018. MongoDB. Retrieved 2019-01-20 from https://www.mongodb.com

[24] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048

[25] Inc. Oracle. 2019. Java Platform, Standard Edition Nashorn User's Guide. Retrieved 2019-02-02 from https://docs.oracle.com/javase/10/nashorn/toc.htm

[26] Ketan Shah, Anirban Mitra, and Dhruv Matani. 2010. An O(1) algorithm for implementing the LFU cache eviction scheme.

[27] Matei Zaharia. 2014. *An Architecture for Fast and General Data Processing on Large Clusters.* Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-12.html